A2 – Scalable Servers

Ben Soer

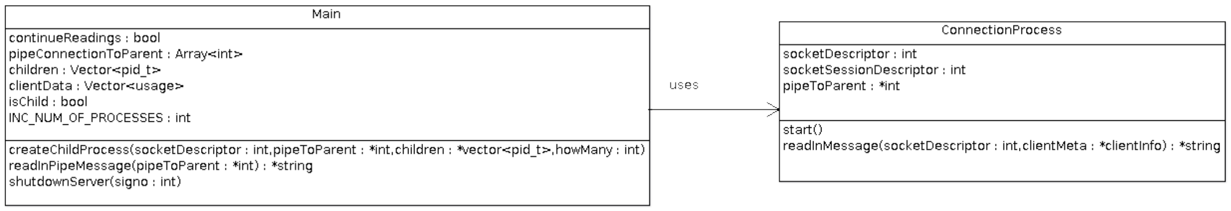
A00843110

Table of Contents

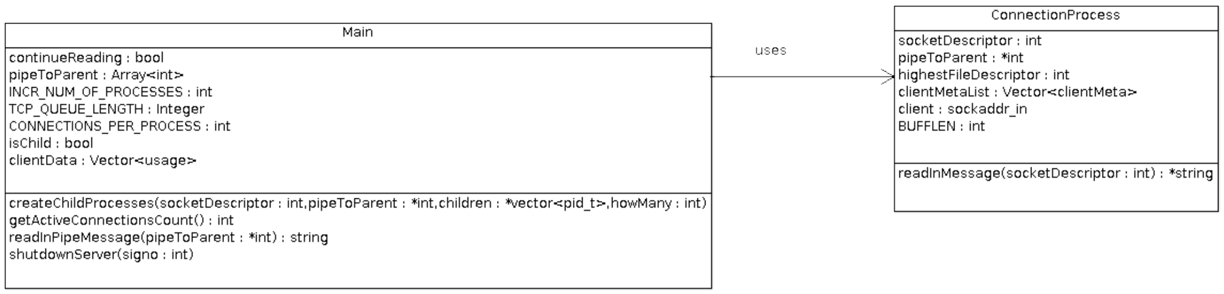
Class Diagrams

You can view all of the original images in the docs/imgs/class folder

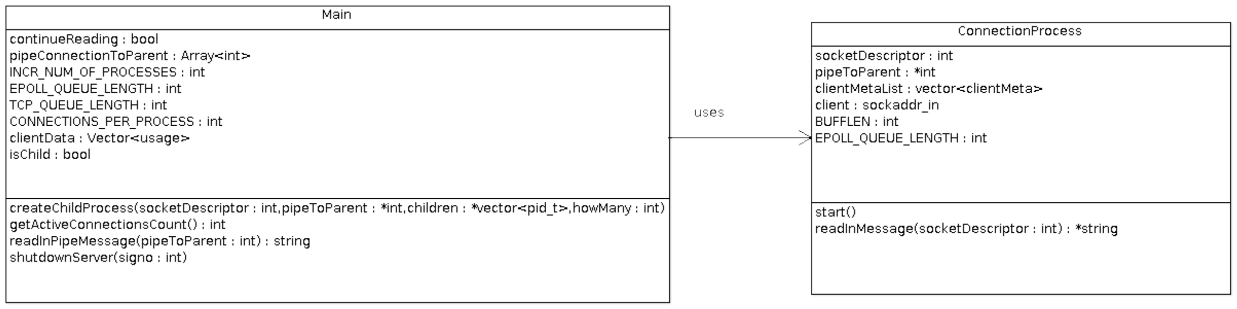
Traditional


Main
continueReadings : bool
pipeConnectionToParent : Array<int>
children : Vector<pid_t>
clientData : Vector<usage>
isChild : bool
INC_NUM_OF_PROCESSES : int
createChildProcess(socketDescriptor : int,pipeToParent : *int,children : *vector<pid_t>,howMany : int)
readInPipeMessage(pipeToParent : *int) : *string
shutdownServer(signo : int)
uses
ConnectionProcess
socketDescriptor : int
socketSessionDescriptor : int
pipeToParent : *int
start()
readInMessage(socketDescriptor : int,clientMeta : *clientInfo) : *string

Select


Main
continueReading : bool
pipeToParent : Array<int>
INCR_NUM_OF_PROCESSES : int
TCP_QUEUE_LENGTH : Integer
CONNECTIONS_PER_PROCESS : int
isChild : bool
clientData : Vector<usage>
createChildProcesses(socketDescriptor : int,pipeToParent : *int,children : *vector<pid_t>,howMany : int)
getActiveConnectionsCount() : int
readInPipeMessage(pipeToParent : *int) : string
shutdownServer(signo : int)
uses
ConnectionProcess
socketDescriptor : int
pipeToParent : *int
highestFileDescriptor : int
clientMetaList : Vector<clientMeta>
client : sockaddr_in
BUFFLEN : int
readInMessage(socketDescriptor : int) : *string

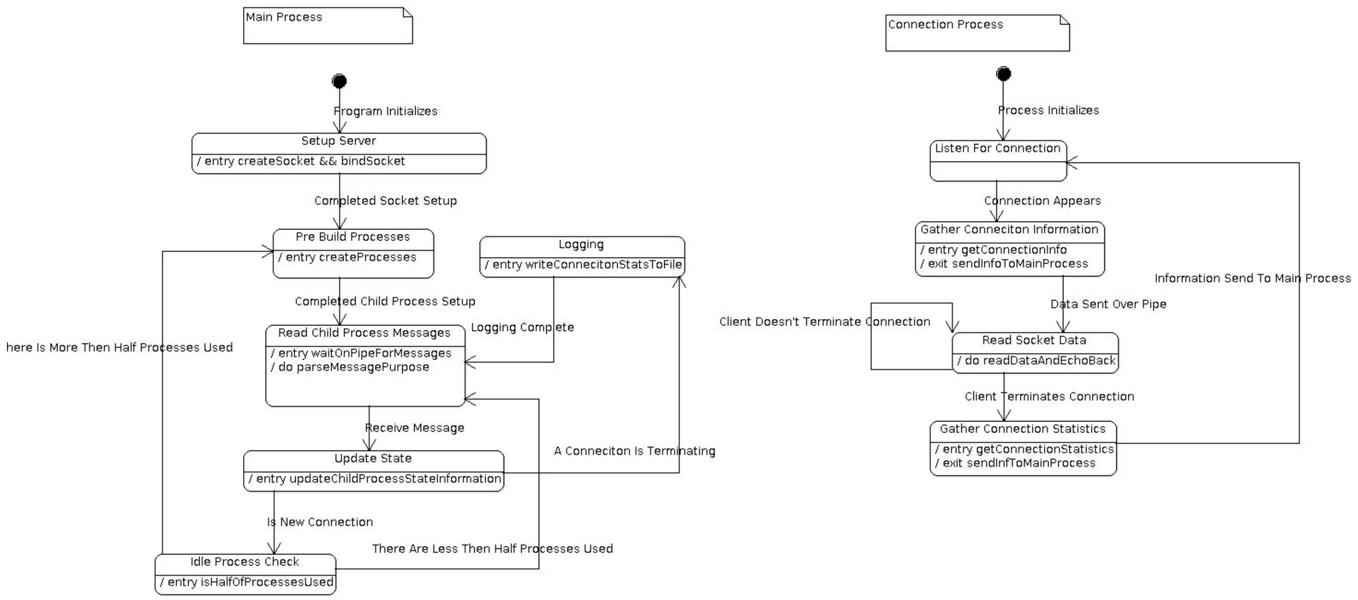
Epoll


Main
continueReading : bool
pipeConnectionToParent : Array<int>
INCR_NUM_OF_PROCESSES : int
EPOLL_QUEUE_LENGTH : int
TCP_QUEUE_LENGTH : int
CONNECTIONS_PER_PROCESS : int
clientData : Vector<usage>
isChild : bool
createChildProcess(socketDescriptor : int,pipeToParent : *int,children : *vector<pid_t>,howMany : int)
getActiveConnectionsCount() : int
readInPipeMessage(pipeToParent : int) : string
shutdownServer(signo : int)
uses
ConnectionProcess
socketDescriptor : int
pipeToParent : *int
clientMetaList : vector<clientMeta>
client : sockaddr_in
BUFFLEN : int
EPOLL_QUEUE_LENGTH : int
start()
readInMessage(socketDescriptor : int) : *string

Finite State Machines

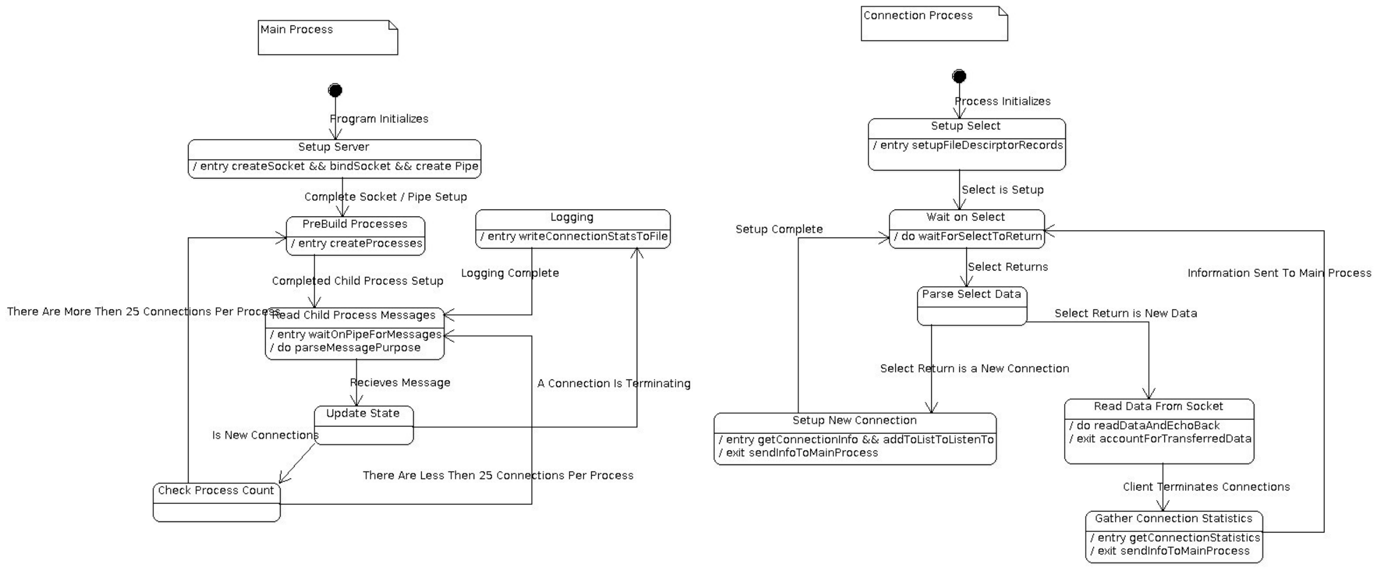
Traditional

See imgs/statechart/TraditionalServerStateMachine.jpg for seperate image


Main Process
Program Initializes
Setup Server
/ entry createSocket && bindSocket
Completed Socket Setup
Pre Build Processes
/ entry createProcesses
Logging
/ entry writeConnecitonStatsToFile
Completed Child Process Setup
Logging Complete
here Is More Then Half Processes Used
Read Child Process Messages
/ entry waitOnPipeForMessages
/ do parseMessagePurpose
Receive Message
A Conneciton Is Terminating
Update State
/ entry updateChildProcessStateInformation
Is New Connection
There Are Less Then Half Processes Used
Idle Process Check
/ entry isHalfOfProcessesUsed
Connection Process
Process Initializes
Listen For Connection
Connection Appears
Gather Conneciton Information
/ entry getConnectionInfo
/ exit sendInfoToMainProcess
Information Send To Main Process
Data Sent Over Pipe
Client Doesn't Terminate Connection
Read Socket Data
/ do readDataAndEchoBack
Client Terminates Connection
Gather Connection Statistics
/ entry getConnectionStatistics
/ exit sendInfToMainProcess

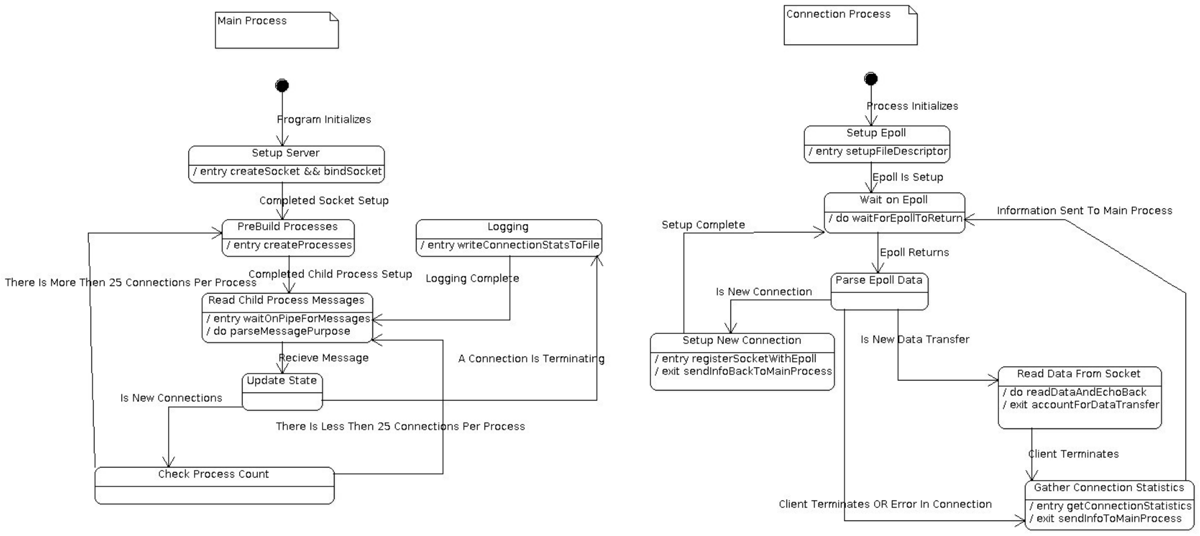
Select

See imgs/staechart/SelectServerStateMachine.jpg for seperate image


Main Process
Program Initializes
Setup Server
/ entry createSocket && bindSocket && create Pipe
Complete Socket / Pipe Setup
PreBuild Processes
/ entry createProcesses
Logging
/ entry writeConnectionStatsToFile
Completed Child Process Setup
Logging Complete
There Are More Then 25 Connections Per Process
Read Child Process Messages
/ entry waitOnPipeForMessages
/ do parseMessagePurpose
Recieves Message
A Connection Is Terminating
Update State
Is New Connections
There Are Less Then 25 Connections Per Process
Check Process Count
Connection Process
Process Initializes
Setup Select
/ entry setupFileDescirptorRecords
Select is Setup
Setup Complete
Wait on Select
/ do waitForSelectToReturn
Select Returns
Information Sent To Main Process
Parse Select Data
Select Return is New Data
Select Return is a New Connection
Setup New Connection
/ entry getConnectionInfo && addToListToListenTo
/ exit sendInfoToMainProcess
Read Data From Socket
/ do readDataAndEchoBack
/ exit accountForTransferredData
Client Terminates Connections
Gather Connection Statistics
/ entry getConnectionStatistics
/ exit sendInfoToMainProcess

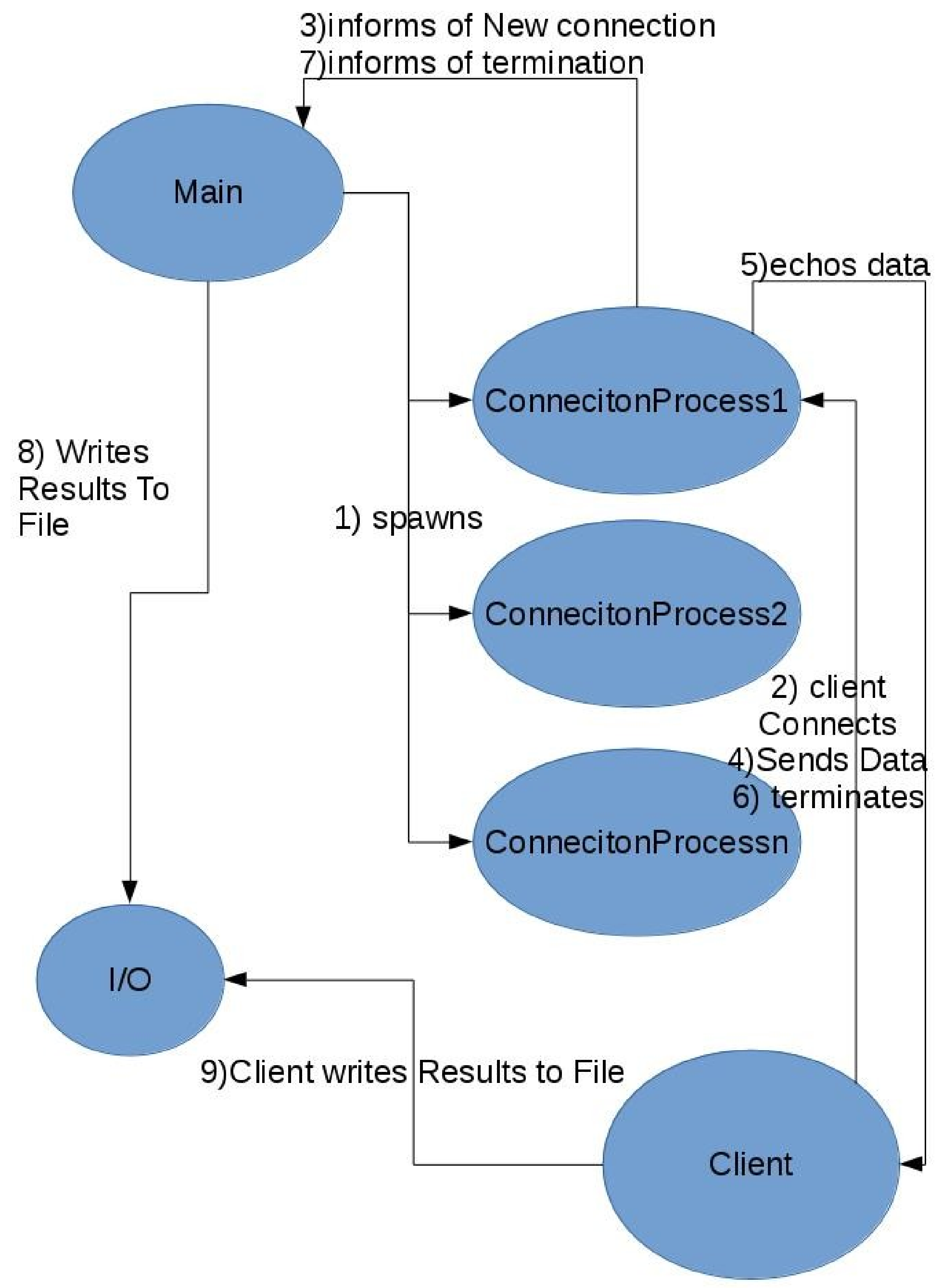
Epoll

See imgs/staechart/EpollServerStateMachine.jpg for seperate image


Main Process
Program Initializes
Setup Server
/ entry createSocket && bindSocket
Completed Socket Setup
PreBuild Processes
/ entry createProcesses
Logging
/ entry writeConnectionStatsToFile
There Is More Then 25 Connections Per Process
Completed Child Process Setup
Logging Complete
Read Child Process Messages
/ entry waitOnPipeForMessages
/ do parseMessagePurpose
Recieve Message
A Connection Is Terminating
Update State
Is New Connections
There Is Less Then 25 Connections Per Process
Check Process Count
Connection Process
Process Initializes
Setup Epoll
/ entry setupFileDescriptor
Epoll Is Setup
Wait on Epoll
/ do waitForEpollToReturn
Information Sent To Main Process
Setup Complete
Epoll Returns
Parse Epoll Data
Is New Connection
Setup New Connection
/ entry registerSocketWithEpoll
/ exit sendInfoBackToMainProcess
Is New Data Transfer
Read Data From Socket
/ do readDataAndEchoBack
/ exit accountForDataTransfer
Client Terminates
Gather Connection Statistics
/ entry getConnectionStatistics
/ exit sendInfoToMainProcess
Client Terminates OR Error In Connection

Data Flow Diagrams

Due to the similar nature of all the the server's designs. The data flow diagram is identical for all of the different servers. They all try to solve the same problem, the same way, except for the tool which they use for listening to and interacting the the sockets. Below is a general data flow of how the servers all function and pass data around

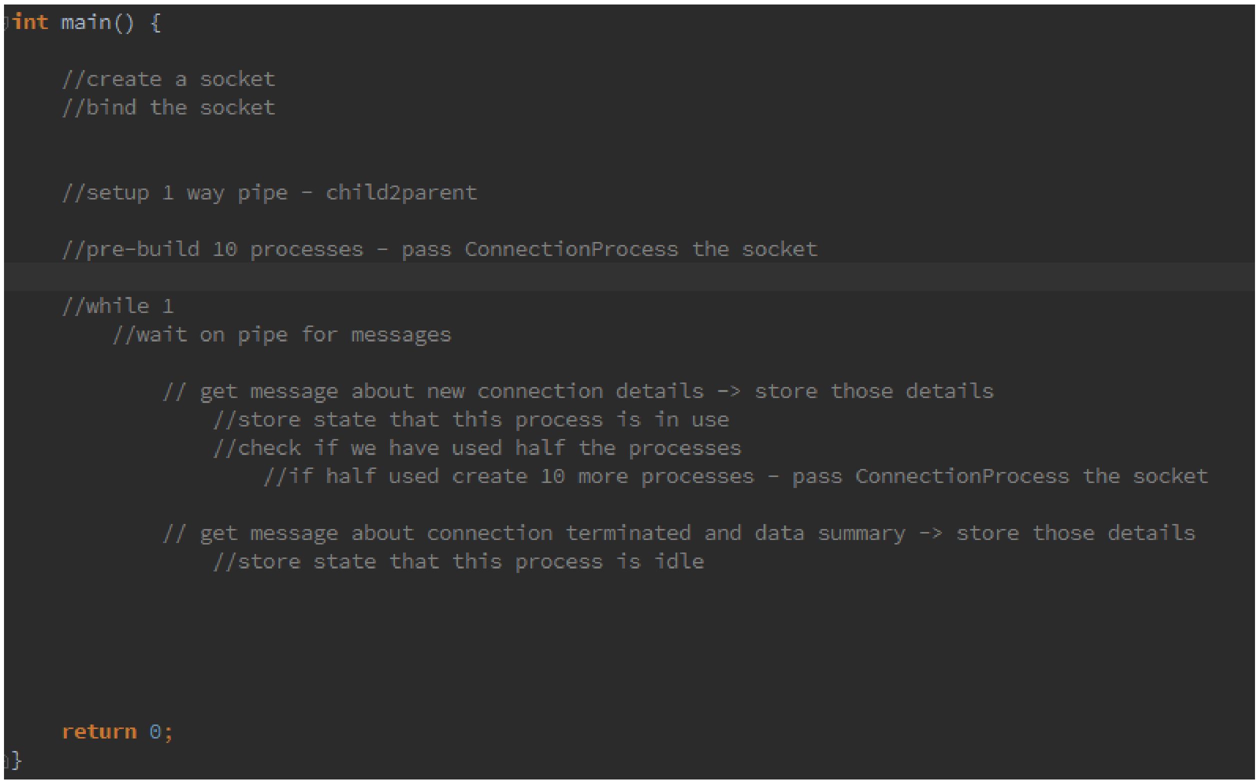

3)informs of New connection
7)informs of termination
Main
5)echos data
ConnecitonProcess1
8) Writes Results To File
1) spawns
ConnecitonProcess2
2) client Connects
4)Sends Data
6) terminates
ConnecitonProcessn
I/O
9)Client writes Results to File
Client

Pseudocode

Traditional

Main.cpp (Program Main Entry)

Main()



```
int main() {

    //create a socket
    //bind the socket


    //setup 1 way pipe - child2parent

    //pre-build 10 processes - pass ConnectionProcess the socket

    //while 1
        //wait on pipe for messages

            // get message about new connection details -> store those details
                //store state that this process is in use
                //check if we have used half the processes
                    //if half used create 10 more processes - pass ConnectionProcess the socket

            // get message about connection terminated and data summary -> store those details
                //store state that this process is idle




    return 0;
}
```

shutdownServer(int signo)

1. Check if caller is child process, if child return immediately
2. If parent get all children and SIGTERM them and then self terminate

*readInPipeMessage(int * pipeToParent)*

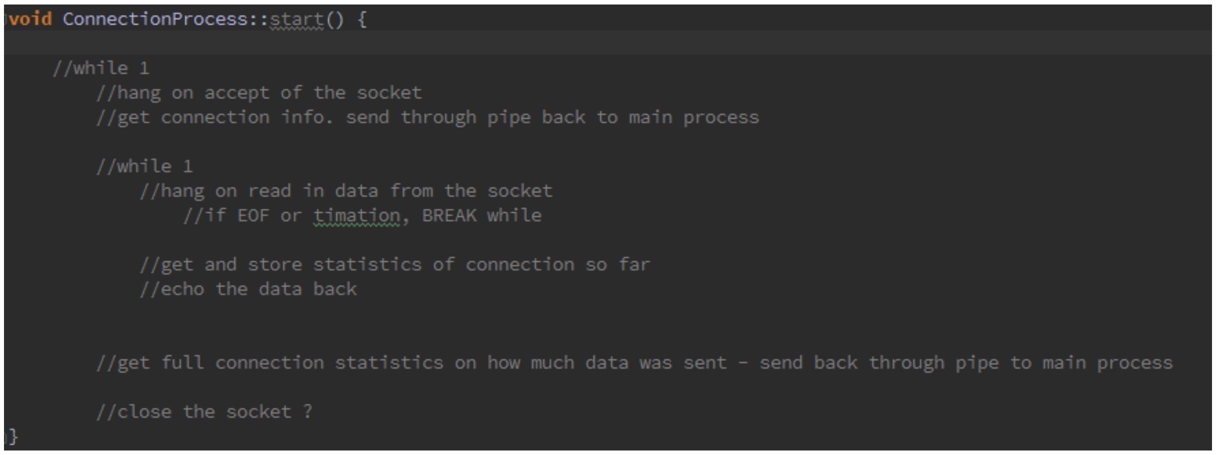
1. While character is not '}' character. Read next character
2. Once find '}' (means end of message). Assemble message and return

*createChildProcesses(int socketDescreiptor, int * pipeToParent, vector<pid_t> * children, unsigned int howMany)*

1. Generate as many child processes as specified by 'howMany' parameter. Pass the pipe to the child process.
2. In the fork, if child, create a ConnectionProcess object, passing the socketDescriptor and pipeToParent
3. In the fork, if child, call start() method of ConnectionProcess
4. In the fork, if parent, add the child process id to the child process records

ConnectionProcess.cpp (Wrapper of Child Process)

Start()



```
void ConnectionProcess::start() {

    //while 1
        //hang on accept of the socket
        //get connection info. send through pipe back to main process

        //while 1
            //hang on read in data from the socket
                //if EOF or timation, BREAK while

            //get and store statistics of connection so far
            //echo the data back


        //get full connection statistics on how much data was sent - send back through pipe to main process

        //close the socket ?
}
```

readInMessage(int socketDescriptor)

1. Read from passed in socketDescriptor
2. If 0 bytes read return nullptr → this tells caller the client dropped connection
3. while '}' is not read in, read in next byte
4. Once found '}' (means end of transfer). Append together message and return

Select

Main.cpp (Program Main Entry)

Main()

1. Setup Socket
2. Setup Pipe For Communicating with Children

3. PreCreate Child Processes
4. While Not Terminated → Wait On Pipe For Messages
    1. If New Connection
        1. Create A Record About The Connection
        2. Check Ratio of How Many Connections There Are To Processes Running. Add More Processes if there are more then 25 connections per process
    2. If Termination
        1. Find Record. Update Record With Complete Information on Transfer
        2. Write Data To File

shutdownServer(int signo)

3. Check if caller is child process, if child return immediately
4. If parent get all children and SIGTERM them and then self terminate

*readInPipeMessage(int * pipeToParent)*

1. While character is not '}' character. Read next character
2. Once find '}' (means end of message). Assemble message and return

getActiveConnectionsCount()

1. Cycle through all connection records. If they are still active (we have no received a termination message from it yet), add them to count.
2. Return the count

*createChildProcesses(int socketDescreiptor, int * pipeToParent, vector<pid_t> * children, unsigned int howMany)*

1. Generate as many child processes as specified by 'howMany' parameter. Pass the pipe to the child process.
2. In the fork, if child, create a ConnectionProcess object, passing the socketDescriptor and pipeToParent
3. In the fork, if child, call start() method of ConnectionProcess
4. In the fork, if parent, add the child process id to the child process records

ConnectionProcess.cpp (Wrapper of Child Process)

start()

1. Setup need structures and arrays for select
2. infinite while loop
    1. Wait for select to return
    2. Check if the socketDescriptor is set, thus meaning we have a new connection request
        1. Accept the Connection
        2. Create a record and send a message back to Main of new connection
        3. Add socketSessionDescriptor to select structures
    3. Check all descriptors for new data
        1. if new data, read in data
            1. updated records about new data message and bytes sent
            2. echo the message back
        2. if client terminates or read shows client terminated
            1. send message to Main with all information about terminated connection
            2. remove connection from select structures

readInMessage(int socketDescriptor)

5. Read from passed in socketDescriptor
6. If 0 bytes read return nullptr →this tells caller the client dropped connection
7. while '}' is not read in, read in next byte
8. Once found '}' (means end of transfer). Append together message and return

Epoll

Main.cpp (Program Main Entry)

Main()

1. Setup Socket
2. Setup Pipe For Communicating with Children
3. PreCreate Child Processes
4. While Not Terminated → Wait On Pipe For Messages
    1. If New Connection
        1. Create A Record About The Connection
        2. Check Ratio of How Many Connections There Are To Processes Running. Add More Processes if there are more then 25 connections per process
    2. If Termination
        1. Find Record. Update Record With Complete Information on Transfer
        2. Write Data To File

shutdownServer(int signo)

1. Check if caller is child process, if child return immediately
2. If parent get all children and SIGTERM them and then self terminate

*readInPipeMessage(int * pipeToParent)*

1. While character is not '}' character. Read next character
2. Once find '}' (means end of message). Assemble message and return

getActiveConnectionsCount()

1. Cycle through all connection records. If they are still active (we have no received a termination message from it yet), add them to count.
2. Return the count

*createChildProcesses(int socketDescreiptor, int * pipeToParent, vector<pid_t> * children, unsigned int howMany)*

1. Generate as many child processes as specified by 'howMany' parameter. Pass the pipe to the child process.
2. In the fork, if child, create a ConnectionProcess object, passing the socketDescriptor and

pipeToParent

3. In the fork, if child, call start() method of ConnectionProcess
4. In the fork, if parent, add the child process id to the child process records

ConnectionProcess.cpp (Wrapper of Child Process)

start()

1. Setup epoll. Create epoll file descriptor. Register socketDescriptor with epoll
2. infinite while loop
3. Wait for epoll to return
4. For loop Through All File Descriptors. For Each File Descriptor:
    1. Check For Errors. If Errors:
        1. Close The Connection
        2. Collect all known accounting information about conneciton and send termination message back to main
        3. continue on next increment of for loop
    2. Check For EPOLLIN. This should never occur. If It Does. Exit(1) process and explode as much as possible
    3. Check If This is A New Connection. If A New Connection:
        1. Accept the Connection.
        2. Create a record about the client
        3. Send A Message Back to Main about the new client
        4. Register the client with Epoll
    4. Otherwise Check For New Data. Call readInMessage passing the file descriptor
    5. If readInMessage returns nullptr:
        1. Client has terminated. Collect information. Send Termination Message to Main
        2. Close connection → deregister connection from epoll
    6. If readInMessage returns message:
        1. Echo back the content
        2. updated records about new data message and bytes sent
        3. delete the returned message

readInMessage(int socketDescriptor)

1. Read from passed in socketDescriptor
2. If 0 bytes read return nullptr →this tells caller the client dropped connection
3. while '}' is not read in, read in next byte
4. Once found '}' (means end of transfer). Append together message and return

Client

Main.cpp

Main()

1. Read in Arguments
2. Setup Socket
3. Establish Connection With Server. Call connectToServer(string host, int port, int socketDescriptor)
4. Setup Epoll To Listen For Replies From Server
5. Infinite While
    1. Send Message. Take Timestamp. Add message number of bytes to running total of data sent from client
    2. Wait For Reply From Epoll
        1. Check For Errors. If Errors Terminate Client
        2. If Not Errors Read In Data
        3. Take Timestamp. Create record of transaction start and finish time and add to connection struct containing requests

shutdownClient(int signo)

1. Stop main's infinite while from running
2. Cycle through list of all transactions and get their times.
    1. Write sendtime, recievetime and deltatime to file
3. Write totalDataSent to file
4. Calculate average RTT of all requests. Write To File

connectToServer(string host, int port, int socketDescriptor)

1. Resolve hostname

2. create socketaddr_in
3. connect to the server. On failure exit client sending error to terminal. On success return